Detecting Hidden Failure Modes in Critical, Embedded Software

Robyn R. Lutz

August 31, 1994

Abstract

This experience report describes a method that has been used successfully to detect hidden failure modes in critical, embedded spacecraft software. The method is an adaptation of an earlier, controversial approach called failure modes and effects analysis. The adapted method was found to be well-suited to identifying latent software design weaknesses involving complex system interactions and dependencies in the two applications described here. This experience may be useful for other high-integrity software systems in which the possibility of hidden failure modes is a major concern.

1. Introduction

Detecting hidden failure modes is a difficult but important problem in the design analysis of critical spacecraft, software. This paper describes the successful application of a design analysis method to the critical software of two spacecraft. The goal was to identify hidden failure modes. The method, an adaptation of earlier failure modes and effects analysis procedures, focuses on the effects on the spacecraft of anomalous inputs and incorrect software activities,

A failure mode is defined to be "the physical or functional manifestation of a failure." A failure is defined to be "an event in which a system or system component does not perform a required function within specified limits" [7].

1.1 The Problem

The software on spacecraft is embedded and distributed on several different flight computers, some of which operate in parallel as redundant, backup units. The possibility of hazardous interactions among the software processes executing on different computers, as well as the complexity of the timing dependencies, make the detection of failure modes difficult. System issues such as storage capacities, noise characteristics, communication protocols, and treacherous operating environments may contribute to software interface failures that elude routine analysis or testing. Similarly, hidden failure modes may involve inadequate software responses to extreme conditions or extreme values. Anomalous hardware behavior, unanticipated states, invalid data, signal saturation, and incorrect triggering of error-recovery

Submitted to 17th International Conference on Software Engineering, Seattle, WA, Apr 24-28, 1995.

software are robustness issues which complicate the production of high-integrity, embedded software.

1.2 Overview of the Method

Software Failure Modes and Effects Analysis (SFMEA) is a method for analyzing the various ways in which software may fail to meet its functional requirements and the consequences of each such failure mode. A SFMEA is typically performed as part of the design analysis process. Ideally, the analysis is performed after a design document exists and before implementation begins.

The goal of the SFMEA is to aid in the production of reliable software. "Defensive design" or designing in protection against hazards is an essential element of efforts to develop robust, highly reliable software [9, 13]. SFMEA's were used in the two applications described here to help reduce the number of failure modes, minimize the effect of the remaining failure modes, and search for unanticipated failure modes.

The SFMEA method was chosen for these applications largely because it contributes to a systems approach to design validation. It focuses on the ways in which software can contribute to the system's reaching a hazardous state. SFMEA's analyze the software response to hardware failures (for example, malfunctioning sensors) or to operator crews that result in bad input data (for example, inappropriate commands or parameter settings). They also analyze the effect on the hardware components of incorrect software actions (for example, a software process issuing wrong commands). SFMEA's are particularly concerned with uncovering hidden dependencies or interactions which could cause the propagation of erroneous data to other software modules.

SFMEA's differ from causal analyses (such as Fault Tree Analysis) in that a SFMEA postulates the existence of bad data, or unexpected behavior and then investigates the effects of that anomaly on the correct functioning of the software module and the system. Whether the data or logic could actually be corrupted in that manner (for example, the arrival of outdated sensor data, or abnormal termination of the software module at a certain point) is not the primary concern. (In fact, judgments as to whether a particular failure scenario is credible on a spacecraft often shift as development and implementation progress.) The focus in the SFMEA is instead on the consequences of incorrect data or inappropriate software activity.

If the effects of bad data or unexpected behavior can be shown to be acceptable, then confidence as to the software's robustness is enhanced. Examples of acceptable effects in the applications described here are that the bad data are rejected by the software logic, or that premature termination of the software module still leaves the subsystem in a consistent state.

If the effects of the bad data or unexpected behavior are shown to be unacceptable, then this information is fed back into the development process or used to develop test cases. Examples of unacceptable effects are that the bad data are used in a control decision resulting in erroneous issuance of commands to spacecraft subsystems, or that an abnormal termination of the software module results in a global variable being updated while the status variable still indicates that no change has been made.

The outline of the rest of the paper is as follows. Section 2 describes related work. Section 3 describes the process used to perform SFMEA's on the critical spacecraft software. Section 4 presents the results found in applying the SFMEA method to those software systems. Section 5 offers some concluding remarks.

2. Related work

Software Failure Modes and Effects Analysis (SFMEA) is an extension of hardware Failure Modes and Effects Analysis (FMEA). Hardware FMEA's have been extensively used and documented. [15]. There is no comparable documented standard for SFMEA 'S, although the usc of SFMEA's is well-documented [5, 17]. The System Safety Analysis Handbook, for example, provides a brief, non-procedural description of SFMEA's ['20].

Past reviews of software failure modes and effects analysis techniques have been mixed. Variants of the SFMEA method have been used successfully in several applications. For example, a technique similar to SFMEA's, called Software Error Effects Analysis (SEEA), was used successfully in the development of the rendezvous and berthing software for the Columbus Free Flyer with the Space Station. For critical software, a SEEA was required [22].

More recently, a paper by McDermid and Pumfrey describes a technique for software safety analysis based on a structural approach to the "imaginative anticipation of hazards" [11]. Unlike SFMEA's, their work concentrates on information flows and develops sets of guide words to prompt consideration of the hypothetical failures. Like SFMEA's, their approach includes table-based analysis that considers the effects of each hypothetical failure, as well as the failure's criticality and likelihood.

The usc of failure modes and effects analysis for software has been criticized because the failure modes for a specific software system, particularly a complex, embedded system, cannot be enumerated. SFMEA's differ from hardware FMEA's in that, while hardware can be reasonably considered to have a limited number of failure modes, software has a much larger and unpredictable number of failure modes. SFMEA's do not offer as high a degree of confidence in the design's correctness as do hardware FMEA's, since SFMEA's cannot provide certain coverage of all failure modes. our experience was that, despite this limitation, SFMEA's were useful in identifying additional failure modes. By detecting hazards not previously recognized during requirements and design analysis, the SFMEA's contributed to the overall robustness of the system.

The SFMEA is not considered to be an adequate validation tool by some software engineers because of its informality. It is a manual rather than an automatic method and depends on the knowledge of the analyst and the accuracy of the documentation. These limitations suggest that for some applications and development environments, SFMEA's are not an appropriate choice. 'The use of SFMEA's described here was chosen to supplement, rather than replace, other requirements and design validation efforts by the development team to produce highly reliable software.

Attempts to automate a process similar to SFMEA's (for example, by expanding a directed-graph fault tree analysis tool so that errors can be introduced and their effects tracked) have not provided a substitute for manual SFMEA's [3, 4]. Because automated

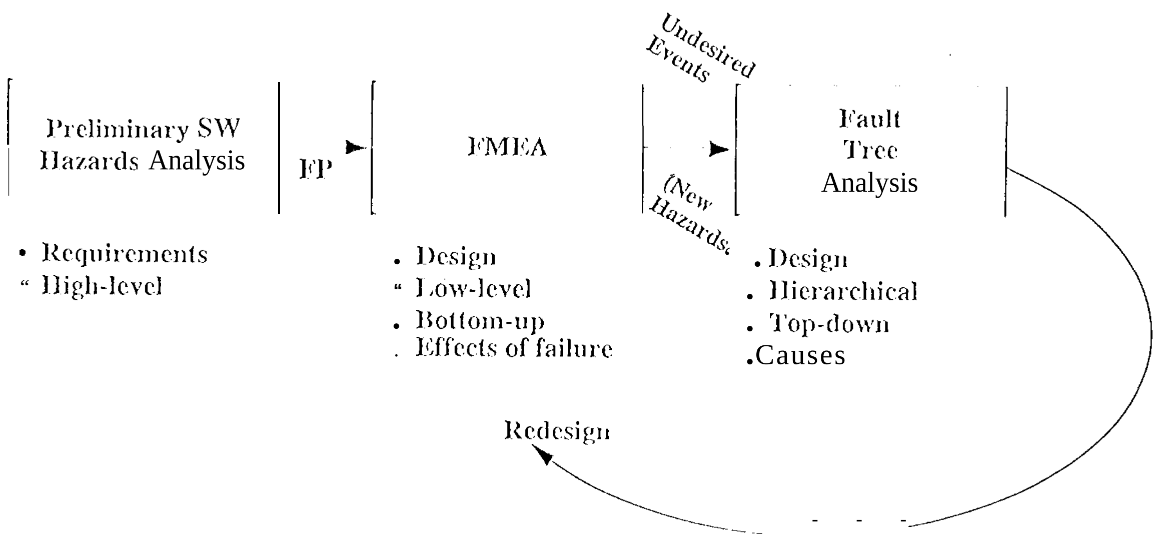

Figure 1: The Role of Failure Modes in Analyzing Hazards

fault-analysis tools require creating a model Of the system as the first step, the results tend to mirror the designer's understanding of the system and so lack the independent validation advantages of the SFMEA. More rigorous state-reachability analyses, though useful, require thorough and time-consuming modelling of the system.

The experience reported here describes two applications in which the problem was to detect hidden failure modes in critical, embedded software. The SFMEA method proved to be a useful and effective tool with which to undertake the analysis needed to solve the problem in these applications. The remaining sections of this paper report on that experience.

A SFMEA has several advantages that are not shared by other validation methods. First, the simplicity of the method makes it easy for developers to review the results and incorporate them ink) sill.)scx]llcllt design iterations or test plans. Secondly, a SFMEA allows independent validation} of the design from the existing documents (rather than requiring the reduction of the existing documentation into a simplified digraph or state model). Thirdly, although all validation methods analyze the correctness of the design in the nominal case, a SFMEA also postulates the existence of bad data and incorrect software activities. The SFMEA considers the effects on the software and the system Of anomalous inputs and anomalous events. A SFMEA thus analyzes the robustness of the software, aswell as its functional correctness.

The relationship between the SFMEA and other static evaluation methods is described in Figures 1 and 2. Figure 1 shows how the results Of a preliminary software hazards analysis in the requirements phase can identify failure modes from which the software must rcc.over. For example, for system-level error recovery, the software must detect and respond appropriately to such problems as a power 10ss, propellant tank overpressure, excessive temperature, interruption of uplink commandability, and loss of downlinked scientific and engineering telemetry. A SFMEA analyzes whether the design of the onboard error-recovery responses is adequate and whether other, unidentified failure modes can mist or be inadvertently caused by the error-recovery software. If further analysis is needed of newly identified failure modes, a faultl-tree analysis can be undertaken. A fault-tree analysis takes a known hazard as its

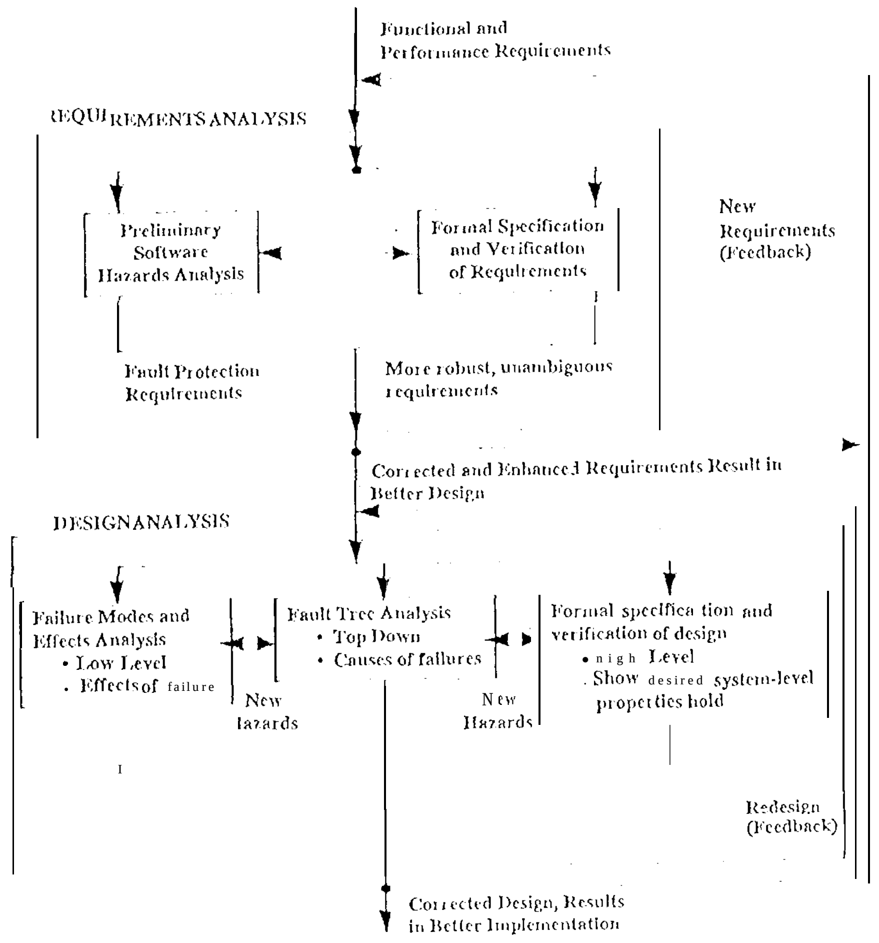
Functional and Performance Requirements
REQUIREMENTS ANALYSIS
Preliminary Software Hazards Analysis
Formal Specification and Verification of Requirements
New Requirements (Feedback)
Fault Protection Requirements
More robust, unambiguous requirements
Corrected and Enhanced Requirements Result in Better Design
DESIGN ANALYSIS
Failure Modes and Effects Analysis
• Low Level
• Effects of failure
Fault Tree Analysis
• Top Down
• Causes of failures
Formal specification and verification of design
• High Level
• Show desired system-level properties hold
New Hazards
New Hazards
Redesign (Feedback)
Corrected Design, Results in Better Implementation

Figure 2: SFMEA and Related Tools

root and winks backwards to determine the possible muses [2].

Figure 2 shows an expanded view of the SFMEA in the context of other analysis tools. SFMEA is a design-analysis method (bottom half of figure) which, based on needs, can be applied alone or in combination with other analysis tools to identify and control failure modes.

3. Applying SFMEA's to Critical Spacecraft Software

SFMEA's were performed on the low-level design of thirteen software processes on Spacecraft A. These software processes are highly critical in that they are the autonomous processes responsible for error recovery onboard the spacecraft. The analyses were performed midway through the development process (after design documentation existed but before code existed). The SFMEA's established a baseline consistency between requirements and design, and were effective in finding significant design errors. Later participation in walkthroughs of the software design on Spacecraft A, audits of the code (tracing the design into the actual software and verifying that known errors were corrected without introducing new errors), and review of software problem and failure reports generated during system testing were all grounded in the earlier SFMEA work. Similar efforts are now underway on the critical error-recovery software processes on Spacecraft B. To date, three SFMEA's have been performed

01) Spacecraft B.

The SFMEA's performed cm Spacecraft A's software used a hardware-like vocabulary ("open," "short," "opposite," "other") to describe the possible defects in the software. This classification was later expanded for Spacecraft B to include a wider range of defects (for example, timing) that have been found to recur in error-recovery software [1 0]. The revised classification offers greater coverage of defects and is consistent with current classifications of errors in software [1 , 12, 14, 1 9].

Another change that was made was to assign criticality and likelihood 1'stings to the effects. This addition was in response to requests from the requirements and design engineers, who use the criticality information to prioritize their redesign and testing needs. The criticality rating is an ordered pair. The first element of the pair refers to a six-tiered classification Of the effect Of the failure on the system (from '510 noticeable impact" to "complete loss of mission.") The second element of the pair classifies the probability of the failure occurring, based 011 experience with similar software ("high," "medium," and '(low.") [16].

Software Failure Modes and Effects Analysis is a design-phase validation tool. Design documentation including textual descriptions and graphical specifications is used in the SFMEA process. Diagrammatic descriptions of the functional flow, such as flow charts of each software module, object-o~iclltc(l data-flow and state diagrams, or Statecharts and Activity Charts, are used as input to the SFMEA Process, if they exist [6, 18].

The following steps were performed for each error-recovery software Process that was analyzed.

1. The normal operation of the subsystem or function to be protected by the software was described. This description was based 011 the requirements and design documentation, the analyst's understanding of the system, and additional explanations from the project personnel, as needed. An example, from Spacecraft A's system-level software that monitors and responds to the loss of a health indicator (a "heartbeat" sent between computers) was a description of how the heartbeat function behaves.

2. The possible functional failures of the subsystem or function to be protected were described. Continuing with the Heartbeat-Loss example introduced above, this step described failures such as "no heartbeat," "heartbeat not updated," "heartbeat updated but garbage," and "heartbeat not synchronized with expected value". Again, the information needed for this step was available in the design documentation and from conversations with the requirements and design engineers.

3. The normal operation of the software in protecting the subsystem or function was described. This step identified how the system-level error-recovery software responded to each of the failures listed above. The information was available from an analysis of the documentation and follow-up discussions with the designers. This step validated the adequacy of the design to accomplish the required recovery and confirmed the SFMEA analyst's understanding of the software.

4. The possible failure modes and effects of the error-recovery software were identified. This step was the crux of the SFMEA, since it analyzed both whether the software

protected against known failures and whether the software could cause additional failures, e.g., by responding to transients or by configuring the spacecraft to a minimally useful safe state when such a reconfiguration was unnecessary. of special concern was the possibility of unexpected interactions among redundant hardware components and computers or among the software processes. For example, in the Heartbeat-Loss example, the SFMEA investigated scenarios in which a failure or apparent failure of the heartbeat would not prompt a correct response, or in which an inappropriate response could create a problem where none existed previously.

A pair of tables was constructed for each software process to assist in the analysis of any possible failures of the software. These were the Data Table and the Events 'J'able. A Data 'J'able involves communication failures. It provides the information needed to analyze data dependencies and software interface errors. An Events Table involves software process failures (where "process" means "the program in execution") [8, 21]. The Events Table provides the information needed to analyze the effects of failures possibly caused by software that fails to function correctly.

The first type of table is the *Data Table.* This table evaluates:

1. the effect of receiving bad input data on the behavior of the process being analyzed.

2. the effect of producing bad output data on the behavior of the processes that use this data.

For each input (data item read or received by the software process) and each output (commands to spacecraft subsystems and updates of data items), each of the following four faults is postulated:

1. Absent Data: Lost or missing messages, absence Of sensor input data, lack of input or output, failure to receive needed data, missing commands, missing updates of data values, data loss due to hardware failures, failure of a software process or sensor to send the data needed for correct functioning of this software module.

2. Incorrect Data: Bad data, flags or variables set to values that don't accurately describe the spacecraft's state or the operating environment, erroneous triggers, limits, deadbands, delay timers; erroneous parameters, wrong commands output, or wrong parameters to the right commands; spurious or unexpected signals.

3. Timing of Data Wrong: Data arrives too late to be used or be ac.curate, or too tally to be used or be accurate, obsolete data is used in control decisions (data age); inadvertent, spurious (unexpected), or transient data.

4. Duplicate Data: Redundant copies of data, data overflow, data saturation.

Each of these four columns is subdivided into two parts: *Description* and *Effect.* The *Description* subcolumn describes the fault as applied to the relevant data item. For example, if the data item is an input called critical-sequence-flag, and the data fault type is "incorrect value," the *Description* subcolumn might state, "flag set to true during non-critical sequence."

The *Effect* subcolumn is a shorthand description of the consequence of the data fault type locally on the data item and more globally on the subsystem and system. In the example given, the entry might state, "Error-recovery response called erroneously." (For ease of reference, entries may also provide a forward reference to a paragraph in the SFMEA that contains further analysis of the item.) In general, the effect of a fault on input data will be either that a state is not updated as it should be, or that the state change is not visible to the software that uses it. The effect of a fault on output data will usually be that other components (software processes or hardware units) lack the information they need to function correctly.

The second type of table is the *Events Table*. This table describes both the local effect of performing an incorrect event on this module's behavior and the global, or end, effect of the incorrect event on other parts of the subsystem and system. For each event that occurs as the process executes, four event fault types are postulated. What constitutes an event depends on the level of detail of the documentation provided, but is usually considered to be a single action (e.g., branch, read, write, output a command).

There are four kinds of event fault types:

1. Halt/Abnormal Termination: Open, stuck, hung, deadlocked at this point (event) in the process.

2. Omission: Event fails to occur but process continues execution; jumps, skips, short.

3. Incorrect Logic/Event: Behavior is wrong, logic is wrong, branch logic is reversed ("greater than or equal " or '(less than and equal"is associated with wrong conditions), wrong assumptions about state, preconditions, "don't cares" aren't truly SO, event (e.g., command issue(I) is wrong to implement the intent or requirement.

4. Timing/Order: Event occurs at wrong time or in wrong order, event occurs too early (premature; system not in proper mode to receive or process it), too late; the sequence of' events is incorrect, an event that must precede another event doesn 't occur as it should; iterative events occur intermittently rather than regularly, events that should occur only once instead occur iteratively.

Each of these columns is subdivided into two parts: *Description* and *Effect*. The *Description* subcolumn describes the fault as applied to the event. For example, if the event is the calculation of a pointer into a table and the event fault is "Incorrect logic," the description might state, "Pointer will be miscalculated."

The *Effect* subcolumn is a shorthand description of the consequence of the event fault type on the relevant event and the failure mode(s) that might result. In the example given, the entry might state, "Pointer points past end of table, effectively bypassing the intended reconfiguration." (As with the Data Table, entries may also provide a forward reference to a, paragraph in the SFMEA that contains further analysis of the item.) In general, the effect of a Halting Failure will be that there is no output from the software response. The possibility that some outputs (e.g., updates of shared variables) occur before the process halts carries a risk of the spacecraft being left in an inconsistent state. The effect of an Omission Failure is often that no output or incorrect output is produced (e.g., wrong time or wrong order). Again, the software may be left in an inconsistent state. Most often, the effect of Incorrect Logic is that the software's behavior is wrong, i.e., it doesn't satisfy the functional requirements or produces wrong; output.

The effect of a Timing/Order Failure is usually that the output doesn't satisfy the timing constraints, required order of commands (e.g., "Instrument must be turned off before replacement heater is turned on"), or data dependencies (e.g., "The flag must be updated before it is used") needed for a correct interface with the other processes that use this software process' output.

The effect of the various failure types included analyses of what the effects would be if the correct design (as presented in the document) were to be implemented incorrectly. That is, the SFMEA considered not only effects of flaws in the design, but effects Of incorrectly implementing a correct design. The goal was to produce more robust software by analyzing what could go wrong, not just whether the current design was flawed.

Using the SFMEA tables, concerns and possibly vulnerable areas were identified. These were documented in sufficient detail so that a reader could determine whether the design or the requirements needed to be changed. Most of the effort of performing a SFMEA was expended here, in reviewing the SFMEA tables to see if the software lacked robustness against failures. During this step, contact with the software designers and requirement engineers was important to distinguish ambiguous/incomplete documentation from design flaws. Including copies of all relevant documentation as well as explicit references to memos, expert opinions, etc., in the final reports encouraged rapid feedback of' the results into the design development process.

The results of each SFMEA were written up in three parts: (I) Documentation inconsistencies/ ambiguities/inaccuracies/ omissions (used both for updating documentation and for validating the code against the design); (2) Issues and concerns (possible unanticipated failure modes or effects, ordered according to Criticality); and (3) the supporting SFMEA tables.

4. Results

Table 1 summarizes six classes of failure modes identified during SFMEA of the two systems. These were hidden failure modes in that they involved unanticipated ways in which the systems could fail to meet their functional requirements. The potential effect of the failure in each case was significant degradation of performance or loss of redundant capabilities. The probability of the failure occurring in each case was low, since the software analyzed is invoked only at the rare and critical times that onboard error recovery is necessary.

Predictably, the close analysis involved in performing the SFMEA's found many other errors, such as documentation inconsistencies and ambiguities as well as discrepancies between the requirement specifications and the design. These problems were described in the final SFMEA report in a separate section. Their identification during the design validation phase saved time and effort later in the integration and system testing phase. However, they are not included in the (Discussion here since they did not lead to failure modes.

Failure modes involving *unexpected interactions among software processes* were identified in both systems. In both cases, this involved software distributed in different processors.

A11 cxamplc is the usc of an outdated valuc from a failed, remote sensor for a key, control (Ice.isiol] in the central processor.

Table] .]] idden Failure Modes Identified Using Software Failure Modes & Effects Analysis

Result	*Spacecraft A*	*Spacecraft B*
1. Unexpected interactions among software	X	x
2. Erroneous invocation of process	x	x
3. Unexpected interactions among redundant units	x	
4. Unexpected propagation of results	x	
5. Unstated assumptions required for correct bel iavior	X	x
6. Timing requirements not always met	X	x

SFMEA's found instances in both systems in which an *erroneous invocation of the software* could occur. In one case, a transient error was able to provoke an unintended error response. In another case, a persistence counter could be reset contrary to the required behavior, thereby omitting necessary activities.

A hidden failure mode on Spacecraft A involved *unexpected interactions between redundant components*, i.e., between the nearly-i dentical copies of the software resident Cm the p rime and backup processors. Required hardware reconfigurations were omitted when a service rou ti nc was invoked by both redundant componen ts within a certain time interval.

A scenario was found 011 Spacecraft A in which *unexpected propagation of results* could occur. During a programmed delay, certain commands to a remote unit were able to be reissued erroneously.

Unstated assumpti ons require d for correct software behavior could indirectly lead to failure modes in three cases. These involved a1 1 assumed but unchecked precondition that is occasionally false (switch settings), a hidden dependency on human intervention (updating flag values following error- recovery activities), and a hidden dependency on other software's undocumented behavior (reinitializing the value of persistence counters and limits).

Finally, several timing issues , particularly *timing requirements not always met* were found by the SFMEA's through ana lysis of the effects of anomal ous situation s and postulated out-of-range data values.

The summary in Table 1 points out one of the kcy advantages of performing SFMEA's o n the applications described here. SFMEA's emphasize a systems approach to software analysis. They examine the software's response to hardware failures and the effect 011 the hardware of software! actions. In real-time, embedded systems, it is these interfaces that have historically produced the persistent errors that remain hidden until system testing or operations [10]. SFMEA's helped uIIc.over these hidden interactions and dependencies on tlIc spacecraft.

5. Conclusion

In the two spacecraft applications reported here, the SFMEA method was effective in detecting hidden failure modes. The SFMEA's focus 011 tracking the effects of anomalous inputs and incorrect software behavior uncovered design flaws and gaps in the software's robustness. Failure modes involving unexpected interactions among software processes, hidden dependen tics, occasionally false assumptions, and unanticipated timing were found. The experience reported here suggests that SFMEA's may be useful for analyzing embedded software in other high-integrity applications in which complex system interfaces, redundancy management issues, and critical timing demands make the possibility of hidden failure modes a concern.

Acknowledgments

The author thanks Steven Tyler, Robert Keston, Yoko Ampo, and Robert Woodhouse for their work 011 SFMEA's and for their valuable suggestions.

The work described in this paper was calv-ice] out at the Jet I'repulsion Laboratory, California Institute of Technology, and was sponsored by the National Aeronautics and Space Administration.

References

[1] E. A. Addy, "A Case Study on isolation of Safety-Critic.al Software," in *Proceedings of the 6th Annual Conference on Computer Assurance*, NIST/IEEE, 1991, PP. 7'5-83.

[2] S. S. Cha, N. G. Leveson, and T. J. Shimeall, "Safety Verification in Murphy Using Fault Tree Analysis, *Proc of the 10th International Conference on Software Engineering*, Apr, 1988, Singapore, pp. 377-386.

[3] *FEAT (Failure Environment Analysis Tool)*, NASA Cosmic #MSC-21873.

[4] *FIRM (Failure Identification and Risk Management Tool)*, Lockheed Engineering and Sciences Co., Cosmic.

[5] J. R. Fragola and J. II'. Spahn, "The Software Error Effects Analyis; A Qualitative Design Tool ," Record, 1973 *IEEE Symposium on Computer Software Reliability*, *110;1)* 73CH 0741-9C, 1973, pp. 90-93.

[6] *The STATEMATE Approach to Complex Systems*, i- Logix.

[7] *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Std 610.12-1990. New York: IEEE, 1990.

[8] 1,. Lamport and N. Lynch, "Distributed Computing Models and Methods," *Formal Models and Semantics, Vol. B, Handbook of Theoretical Computer Science,* Elsevier, 1990.

[9] N. Leveson, "Software Safety in Embedded Computer Systems, *Communications of the ACM*, Feb, *1991,* Vol. 34, No. 2, pp. 35-46.

[1C R. Lutz, "Analyzing Software Requirements Errors in Safety-Critic.al, Embedded Systems," *Proc of the IEEE International Symposium on Requirements Engineering*, Jan 4-6, 1993, San Diego, CA, pp. 126-133.

[11 J. A. McDermid and D. J. Pumfrey, "A Development of Hazard Analysis To Aid Software Design," *Proc of COMPASS* '94, Jun 27-30, 1994, Gaithersburg, MD, pp. 17-25.

[12 T. Nakajo and H. Kume, "A Case history Analysis of Software Error Cause-ltffcct Relationship," *IEEE Transactions on Software* Engineering, 17, 8, Aug 1991, pp. 830-838.

[13] P. G. Neumann, "The Computer-Related Risk of the Year: Weak Links and Correlated Events," in *Proc 6th Annual Conj on Computer Assurance.* NIST/IEEE, 1991, pp. 5-8.

[14] T. J. Ostrand and E. J. Weyuker, "Collecting and Categorizing Software Error Data in an Industrial Environment," *The Journal of Systems and Software*, *4,* 1984, *pp. 289-300.*

[15] Procedures for Performing a Failure Mode, Effects and Criticality Analysis, MIL-STD-1629A, 24 Nov 1980.

[16] Project Reliability Group, *Reliability Analyses Handbook*, Jet Propulsion laboratory 1)-5703, July, 1990.

[17] 1). J. Reifer, "Software Failure Modes and Effects Analysis, " *IEEE Transactions on Reliability*, *vol. R-28, No. 3,* Aug 1979, PP. 247-249.

[18] J. Rumbaugh, et al., *Object-Oriented Modeling and Design*, Prentice 11 all, 1991.

[19] R. W. Selby and V. R. Basili, "Analyzing Error-Prone System Structure," *IEEE Transactions on Software Engineering*, *17,* 2, Feb 1991, pp. 141-152.

[20] System Safety Society, *System Safety Analysis Handbook*, July, 1993.

[21] A. S. Tanenbaum, *Modern Operating Systems*, Prentice-Hall, 1992.

[22] J. Wunram, "A Strategy for Identification and Development of Safety Critical Software Embedded in Complex Space Systems," IAA 90-557, pp. 35-5].